Afshin Jamali

SE 450

3/8/14

TRAFFIC SIMULATOR REPORT

**Time Summary:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Week** | **1** | **2** | **3** | **4** | **Total** |
| **Design** | 1 | 7 | 10 | 15 | 33 |
| **Code** | 0 | 3 | 2 | 30 | 35 |
| **Bigbug** | 0 | 0 | 4 | 10 | 14 |

82 Hours

**Notes on patterns:**

The design patterns used in my project are Singleton, Factory, Strategy, Observer, and State pattern.

The first problem was updating the GUI after each update for classes Car, Road, and Light. The change had to be made so that all were updated at once, which led to the decision of using the Observer pattern. The classes involved are the TimeServerLinked and Observable classes. To implement, I created the road segments, intersections, lights, connected the roads and lights, of course called the animator builder to draw all the necessary pieces. Finally, I added the animator object to the time server which extends the observable class.

But rather than adding and removing observers in classes or methods one by one, and having observer method calls all over the place, the solution was to create a single object and enqueue all other objects mentioned plus classes Intersection, Source, and Animator . This meant that all classes use one object which is the TimeServerLinked class, which led to the decision of using the Singleton pattern. I spent a lot of hours just trying to wrap my head around this concept. In tackling the Time Server problem, I studied the examples in lecture 7 to see how a Time Server works and then I began modifying and creating interfaces, methods, and so on, and as it turns out it's quite a nice way of getting the gears in motion without the need of the forloop method because each of the Agents enqueue themselves to the time server which then is responsible for signaling changes to the observer. The time server has nodes which store the objects, and then dequeues when next is called along with the method call to run that Agent. The classes involved are TimeServerLinked, Car, Source, Road, Light, Agent, and Intersection.

A problem that arose midway thru design and implementation was how to get distance to obstacle if a car is on a road or on an intersection. The solution was to use strategy so when a call is made to find the distance to obstacle of its current road, return the obstacle position depending on if the car is within an intersection or on a road. The classes involved are

For the menu we had to use the popupUI from homework 3. So it was clear and the solution was already provided, which was the Factory pattern. In my case I used the UIFactory to create PopupUI object through the UI Interface. The classes involved are UI, PopupUI, UIError, UIFactory, UIFormBuilder, UIFormTest, UIMenu, UIMenuAction, UIMenuBuilder.

One solution was clear from the beginning, and that is the State pattern, which I used for the Intersection class to determine the color of the Light depending on weather the state is true or false. To Accomplish this I created enum classes for both LightColor and Direction. The latter was to specify the light state for eastwest and northsouth lights. If the state of light was false, the color would be red, otherwise green or yellow.

**Successes and Failures:**

Early on in the project, I realized that there were many elements involved such as patterns, and classes. I had to focus my attention on each task before I moved on to the next. One of the biggest challenges I had to face was how to implement a car stopping or not at an intersection if the color is yellow. I knew what the requirement was, but implementing it was a little tricky. I realized that I needed to use some kind of state pattern to let the car object know if it was supposed to stop at the light. So when the car is within breaking distance of the yellow light,

Another big challenge was the behavior of cars within an intersection. I could not see how the cars were move, if at all. I had to make sure that logically the cars weren’t running into each other, and also, that the cars wouldn’t climb on top of one another on the way out of the intersection and on to the next road. So I included the distance to carback method to apply to the light class as well. Then I added a check feature to not update the position of a car if all the cars occupy the road.

One more slightly challenging puzzle to solve was creating the road segments and ensuring that everything connected well.

If I could think of a failure, it would be that when I increase the road length to a significant amount, like say, 1000m, the cars appear to infuse themselves like cells in a body. Vice versa, if I reduce road length to say, 20m, there appears a huge gap between cars. I still haven’t figured out how to normalize that.